

前言

展示课后的改进

展示课的时候因为阴影不太正确，老师指出上帝视角下，无法实现全局阴影，需要改进。经过一段时间的摸索，现实时阴影实现效果如下。



图形学大作业报告

The Goblet of Fire_Proposal

一、项目要求

简单三维建模及真实感绘制

基本要求：

- 基于OpenGL，具有基本体素的建模表达能力
- 具有基本三维网格导入导出功能
- 具有基本材质、纹理的显示和编辑能力
- 具有基本几何变换功能（旋转、平移、缩放等）
- 基本光照明模型要求，并实现基本的光源编辑
- 能对建模后场景进行漫游如Zoom In/Out, Pan, Orbit, Zoom To Fit等观察功能。

本项目额外实现：

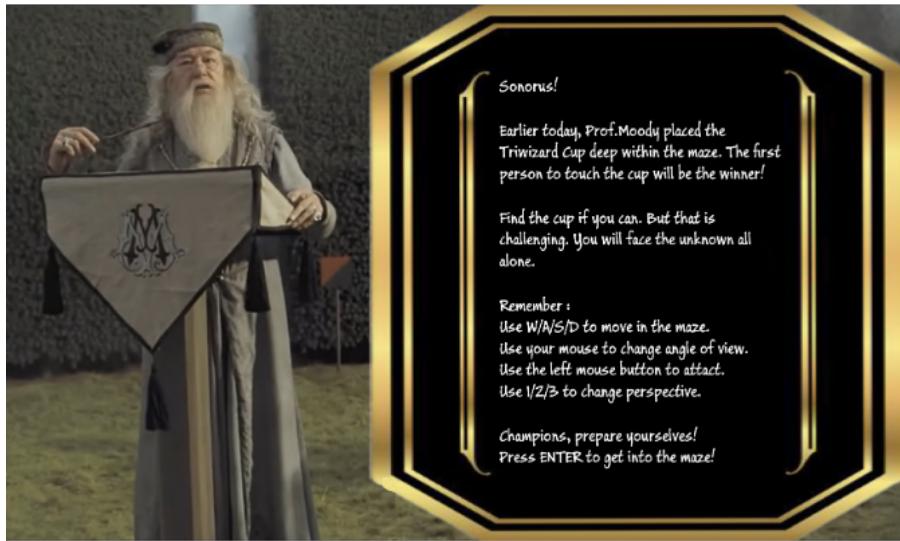
- 实时碰撞检测
- 光照模型细化，支持多种光源（平行光、点光源等）
- 实现PCF实时阴影、HDR渲染等
- 实现法向贴图和位移贴图

二、游戏玩法

执行可执行文件。进入游戏初始界面



界面提示输入 **ENTER** 键进入游戏



为方便玩家进行游玩，在正式进入游戏前，屏幕会出现游戏玩法教程，并提示输入ENTER键正式进入游戏。

玩家将扮演哈利波特三人组，在藤蔓迷宫中找寻隐藏的火焰杯，并使用魔法攻击，击败途中后遇见的不同怪物。



哈利波特三人组



怪物

游戏玩法：

- 玩家通过W/A/S/D键控制角色在地面上移动
- 鼠标控制方向，其中点击鼠标左键可以发射魔法
- 按住左侧shift按键可以实现人物加速
- 键盘上方1/2/3数字键切换视角（包括第一人称视角、第三人称视角和上帝漫游视角）
- Q键开启平行光照、U键开启阴影效果，I键关闭阴影效果，ESC键退出游戏等

三、实现技术

为方便实现炫酷的光照模型等效果，本次项目使用OpenGL中的GLFW应用框架。

3.1 导入模型&&载入地图

导入模型(obj+mtl)

本次实验中我们使用到的复杂模型来源于<https://free3d.com/> 和<https://www.turbosquid.com/>。两个网站上的免费素材，采用obj文件 + mtl文件的模型加纹理数据。由于我们之前对于mtl文件的格式并不了解，在想助教征求同意后我们决定采用glfw库中提供的model库进行模型的导入和贴图。（存放在resource/obj文件夹下）

Model库封装得非常简洁，使用起来也比较方便。但这也导致一些obj文件中的底层信息被忽略，比如在碰撞检测部分就需要用到模型的x、y、z轴范围，此部分的数据处理需要单独实现。（模型范围读取代码存放在resource/obj/getObjSize文件夹下）

载入地图模块

由于本次实验是基于迷宫的游戏，因此迷宫地图的设计十分重要，为了方便设计者更好、更快捷的设计地图，我们设计可读入的地图格式（存放在resource/map文件夹中）。

```
32 32
1011111111111111111111111111111111111111111111111111111111111111
1010100000000000100000000000000000001
101010111111011011011111111101
10000000000010010101001000000000101
1111111111010110110101111111101
100000010010101100101000000001101
10110111011010100110111111101001
101000000000101011000000000001101
1011111101101001010111111010101
101000000111111011010100000111101
10101111000000001001010110000101
101011010111111111110101111110101
10101000010100000000011101110001
101011111101003000000000000011111
101000000001000000000110101000001
101111111100040003010101011111
1010000000003002000000101000001
1010111011110000400011111111111
1110101111010000000000000000000001
100000000001000000000011111111101
1011111011111111011010000000101
1010001000000000010010101110101
1010110111111111111101010000101
10000100000000000100000101011101
11111101111011110101110101000001
10000101001010010101010101111111
10111110101101101010101011110000
1010000100000101010101010100000111
1010110101111100010101010111110001
101111010001111101010100011101
10000001000100000001000101000001
11111111111111111111111111111111
```

- 3和4表示不同种类的怪兽
- 2代表该位置为火焰杯

- 1代表该位置存在方块

- 0代表无物体

具体代码如下，将地图文件读入，转化为游戏中每个方块的位置，并存入 `boxPosition` 数组，用于地图绘制。

```
void CreateMap(const std::string filename)
{
    std::ifstream in(filename);
    std::string s, temp;
    int i = -1, length = 0, width = 0;
    while (getline(in, s))
    {
        if (i == -1)
        {
            std::istringstream instr(s);
            instr >> length >> width;
        }
        else
        {
            int temp;
            std::istringstream instr(s);
            for (int k = 0; k < width; k++)
            {
                instr >> temp;
                if (temp == 1)
                    boxPosition.push_back(glm::vec3(2 * i - 1, 0.0, 2 * k - 1));
            }
            i++;
            if (i == length) break;
        }
    }
}
```

绘制地图效果：



3.2 摄像机系统

在glfw库的camera.h头文件的基础上，改进了摄像机系统，在漫游、视角转换、位置变换等功能的基础上，增添根据三维方向信息，在xz二维平面上的移动，以及根据外设输入信号修改移动速度等功能，并在此基础上实现三种不同视角。

```

void ProcessKeyboard(Camera_Movement direction, float deltaTime, unsigned int viewType)
{
    float velocity = MovementSpeed * deltaTime;
    glm::vec3 FrontIn2D = glm::normalize(glm::vec3(Front.x, 0.0f, Front.z));
    glm::vec3 RightIn2D = glm::normalize(glm::vec3(Right.x, 0.0f, Right.z));
    if (direction == FORWARD)
    {
        if (viewType == FPV || viewType == TPV)
        {
            Position += FrontIn2D * velocity;
            if (Position.y < -1.0f) Position.y = -1.0f;
        }
        else if (viewType == GPV)
        {
            Position += Front * velocity;
            if (Position.y < 0.0f) Position.y = 0.0f;
        }
    }
    if (direction == BACKWARD)
    {
        if (viewType == FPV || viewType == TPV)
        {
            Position -= FrontIn2D * velocity;
            if (Position.y < -1.0f) Position.y = -1.0f;
        }
        else if (viewType == GPV)
        {
            Position -= Front * velocity;
            if (Position.y < 0.0f) Position.y = 0.0f;
        }
    }
    if (direction == LEFT)
    {
        if (viewType == FPV || viewType == TPV)
        {
            Position -= RightIn2D * velocity;
            if (Position.y < -1.0f) Position.y = -1.0f;
        }
        else if (viewType == GPV)
        {
            Position -= Right * velocity;
            if (Position.y < 0.0f) Position.y = 0.0f;
        }
    }
    if (direction == RIGHT)
    {
        if (viewType == FPV || viewType == TPV)
        {
            Position += RightIn2D * velocity;
            if (Position.y < -1.0f) Position.y = -1.0f;
        }
        else if (viewType == GPV)
        {
            Position += Right * velocity;
            if (Position.y < 0.0f) Position.y = 0.0f;
        }
    }
}

```

```
}
```

3.3 纹理显示和编辑

在项目中，我们使用`loadTexture`函数来读取纹理

```
unsigned int loadTexture(char const* path)
{
    // Get textureID
    unsigned int textureID;
    glGenTextures(1, &textureID);

    // use stbi_load to load texture pic
    int width, height, nrComponents;
    unsigned char* data = stbi_load(path, &width, &height, &nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

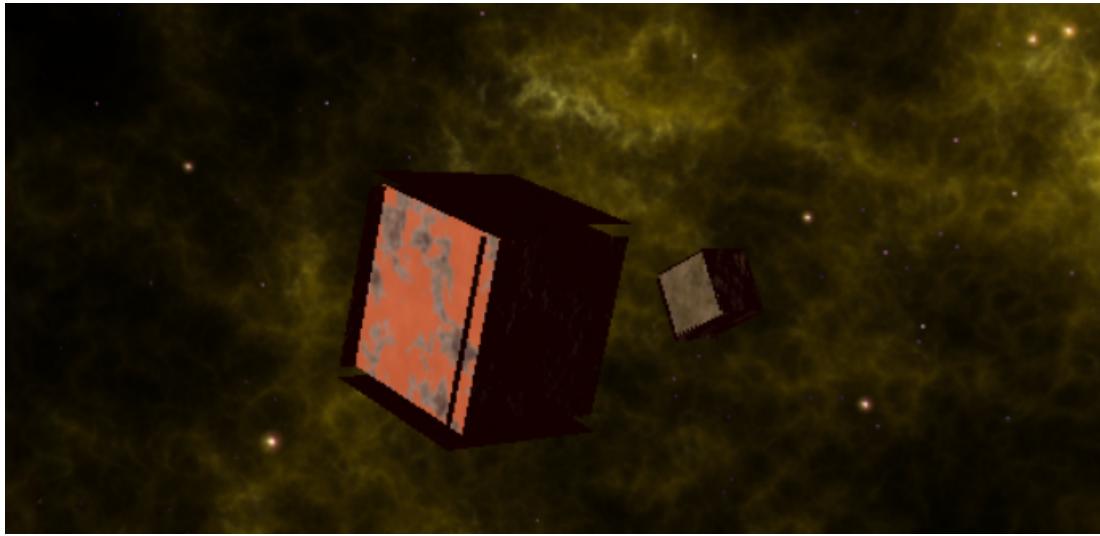
        // Bind texture to the buffer and set data
        glBindTexture(GL_TEXTURE_2D, textureID);
        glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
        GL_UNSIGNED_BYTE, data);
        // Generate mipmap
        glGenerateMipmap(GL_TEXTURE_2D);
        //Set the behavior when scaling
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        // Free the data
        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at path: " << path << std::endl;
        stbi_image_free(data);
    }

    return textureID;
}
```

读取纹理获得相应的`textureID`后，激活纹理即可使用

```
glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, TextureID);
```

本次实验使用了**diffuse**贴图，**specular**贴图，**normal**贴图，**displacement**贴图。前两种广泛使用，而后两种贴图需要在法向空间中计算(但因出现较难解决的问题，故后面舍弃)。



3.4 Phong Shading&&多光照模型

光照采用冯氏光照模型(Phong Lighting Model)。冯氏光照模型的主要结构由3个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。并且实现了可拓展的光照模型（可以添加任意平行光和点光源）。

对应的fs文件如下：

计算点光源

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance +
    light.quadratic * (distance * distance));
    // combine results
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse,
    TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular,
    TexCoords));
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

计算点光源的时候考虑diffuse光和specular光，使用额外的specular贴图和diffuse贴图，并且考虑光线的衰减。

计算平行光

```

vec3 calcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    float diff = max(dot(normal, lightDir), 0.0);
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse,
    TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular,
    TexCoords));
    return (ambient + diffuse + specular);
}

```

计算平行光与点光源最大的不同在于，`lightDir`为固定方向，与物体位置无关，且没有光线的衰减。

合成

```

void main()
{
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 result=vec3(0.0,0.0,0.0);
    if(dir)
    {
        result = calcDirLight(dirLight, norm, viewDir);
    }
    for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
    FragColor = vec4(result, 1.0);
}

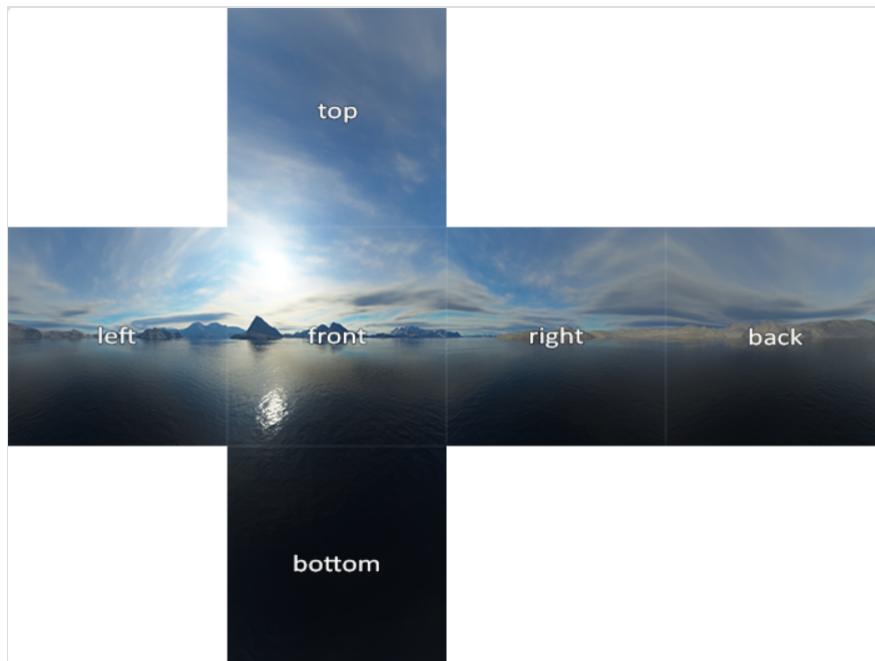
```

将所有的平行光和点光源加和，合成所有的光即可。

3.5 天空盒

我们可以使用立方体贴图来搭建天空盒。天空盒是一个包含了整个场景的立方体，它包含周围环境的6个图像，让玩家以为他处在一个比实际大得多的环境当中。

通常天空盒一共有六张图片(`top, bottom, front, back, right, left`)组成，如下所示。



具体实现代码如下：

```
unsigned int CreateSkybox(int i)
{
    std::vector<std::string> faces;
    if (i == 0)
    {
        faces.push_back(std::string("resource/skybox/skybox3/left.jpg"));
        faces.push_back(std::string("resource/skybox/skybox3/right.jpg"));
        faces.push_back(std::string("resource/skybox/skybox3/top.jpg"));
        faces.push_back(std::string("resource/skybox/skybox3/bottom.jpg"));
        faces.push_back(std::string("resource/skybox/skybox3/front.jpg"));
        faces.push_back(std::string("resource/skybox/skybox3/back.jpg"));
    }
    else if (i == 1)
    {
        faces.push_back(std::string("resource/skybox/right.jpg"));
        faces.push_back(std::string("resource/skybox/left.jpg"));
        faces.push_back(std::string("resource/skybox/top.jpg"));
        faces.push_back(std::string("resource/skybox/bottom.jpg"));
        faces.push_back(std::string("resource/skybox/front.jpg"));
        faces.push_back(std::string("resource/skybox/back.jpg"));
    }
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char* data = stbi_load(faces[i].c_str(), &width, &height,
&nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width,
height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
    }
}
```

```

    else
    {
        std::cout << "Cubemap texture failed to load at path: " << faces[i]
        << std::endl;
        stbi_image_free(data);
    }
}

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

return textureID;
}

```

与其他纹理贴图不同，进行立方体贴图时候，需要调用6次 `glTexImage2D` 函数，并用下面参数指定我们应在哪个面进行

<code>GL_TEXTURE_CUBE_MAP_POSITIVE_X</code>	右
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_X</code>	左
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Y</code>	上
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Y</code>	下
<code>GL_TEXTURE_CUBE_MAP_POSITIVE_Z</code>	后
<code>GL_TEXTURE_CUBE_MAP_NEGATIVE_Z</code>	前

绑定方式使用 `GL_TEXTURE_CUBE_MAP`

```
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

天空盒使用的vs, fs文件如下，顶点着色器中将输入的位置向量作为输出给片段着色器的纹理坐标。片段着色器会将它作为输入来采样 `samplerCube`

```

#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}

```

```

#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

```

```

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}

```

3.6 多视角实现

项目中实现了第一人称、第三人称和上帝（漫游）视角，三种观察模式。

视角模式	是否绘制模型	是否产生碰撞	移动方式
第一人称	否	是	禁止y轴移动
第三人称	是	是	禁止y轴移动
上帝视角	是	否	无限制

借助修改后的camera类，维持两个camera的对象分别表示摄像机的位置以及主角模型所在的位置。其中所有的碰撞、模型绘制均按照主角模型的位置信息检测，而观察位置则按照摄像机的位置信息。即模型表示移动、攻击等信息，摄像机表示视角信息。

- 在第一人称下，二者保持时刻统一

```

if (view_type == FPV)
{
    camera.Position = myPos.Position + glm::vec3(0.0f, 1.5f, 0.0f);
    myPos.Yaw = camera.Yaw;
    myPos.Front = camera.Front;
    myPos.Right = camera.Right;
    myPos.Up = camera.Up;
}

```



- 在第三人称下，摄像机位置保持在模型实现反方向一定距离处，并且模型会随着视角的变化而转动。

```

else if (view_type == TPV)
{
    camera.Position = myPos.Position + glm::vec3(0.0f, 1.8f, 0.0f) -
glm::normalize(glm::vec3(myPos.Front.x, 0.0f, myPos.Front.z)) * 1.2f;
    myPos.Yaw = camera.Yaw;
    myPos.Front = camera.Front;
    myPos.Right = camera.Right;
    myPos.Up = camera.Up;
}
//模型跟随视角旋转
HarryModel = glm::rotate(HarryModel, myPos.Yaw / 57.0f, glm::vec3(0.0f, 0.0f,
-1.0f));

```



在上帝视角下，键盘的移动将控制摄像机的位置，而模型位置保持不变，同时开放y轴移动，且由于碰撞检测是根据模型位置决定，所以摄像机的移动不会限制。最后当玩家从上帝视角切换回第一人称或第三人称视角时，摄像机的位置在此被上述两种关系约束，保持在追随模型位置的状态。



3.7 碰撞检测(AABB)

碰撞检测在我们的游戏中非常重要，迷宫移动时需要用到碰撞检测，进行魔法攻击时也需要计算粒子和物体是否发生碰撞。我们采用的是AABB碰撞检测方法，将模型看成是包裹坐标位置的三维长方体，与其他的长方体的边缘坐标比较，以一个长方体为主，分别判断八个顶点是否位于另一个长方体内部，如果是，则说明发生了碰撞，轮询检测其中距离边界最近的轴坐标，将其移动到外部，之后继续处理剩下的轴坐标，直到该物体完全位于另一个物体的外部；如果不是，则说明没有发生碰撞。

```
if (leftEdge > boxPosition[i].x - BoxWidth / 2 && leftEdge < boxPosition[i].x + BoxWidth / 2 && frontEdge > boxPosition[i].z - BoxWidth / 2 && frontEdge < boxPosition[i].z + BoxWidth / 2)
{
    float a = leftEdge - (boxPosition[i].x - BoxWidth / 2);
    float b = (boxPosition[i].x + BoxWidth / 2) - leftEdge;
    float c = frontEdge - (boxPosition[i].z - BoxWidth / 2);
    float d = (boxPosition[i].z + BoxWidth / 2) - frontEdge;
    float min = (a < b) ? (a < c ? a : c) : (b < c ? b : c);
    min = (min < d) ? min : d;
    if (min == a)
        position.x = boxPosition[i].x - BoxWidth / 2 - eps + radius;
    if (min == b)
        position.x = boxPosition[i].x + BoxWidth / 2 + eps + radius;
    if (min == c)
        position.z = boxPosition[i].z - BoxWidth / 2 - eps + radius;
    if (min == d)
        position.z = boxPosition[i].z + BoxWidth / 2 + eps + radius;
    return false;
}
```

我们在游戏中实现了主角模型与墙体、怪物的碰撞检测，魔法粒子与墙体和怪物的碰撞检测

3.8 粒子效果

项目中共定义两种粒子particle和magic

```
struct Particle {  
    std::vector<glm::vec3> pos;  
    std::vector<glm::vec3> speed;  
    std::vector<glm::vec3> color;  
    int lifespan;  
};
```

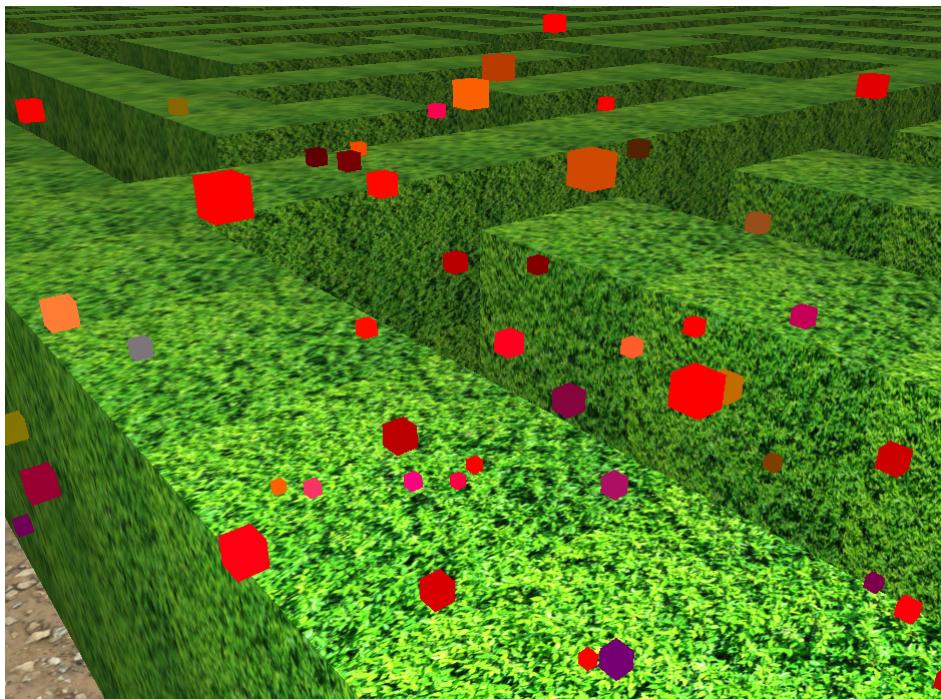
```
struct Magic {  
    std::vector<glm::vec3> pos;  
    glm::vec3 speed;  
    glm::vec3 color;  
    int lifespan;  
    bool disappear;  
};
```

- pos 变量代表粒子团中每个粒子的空间位置
- speed代表粒子的运动速度
- color代表每个粒子的颜色
- lifespan代表粒子的生命周期，超过一定的周期粒子会自动消亡
- Magic类中的 disappear 代表发射粒子是否碰撞到物体，如果碰到物体或超过生命周期，Magic粒子会转变成Particle粒子

下面分别展示Magic粒子（类似子弹），从当前相机位置向前方发射



和Particle粒子（炸开的效果）



具体实现如下，利用随机来实现粒子四处飞散的效果

创建Magic粒子和Particle粒子

```

struct Magic initMagic(glm::vec3 position, GLuint lifespan, glm::vec3 front,
glm::vec3 color, float eps)
struct Particle initParticle(glm::vec3 position, GLuint lifespan, int num,
glm::vec3 color, float eps)
{
    int ParticleNumber = num;
    Particle p;
    for (int i = 0; i < ParticleNumber; i++)
    {
        p.pos.push_back(position);
        p.speed.push_back(glm::vec3(0.00f, 0.0f, 0.00f));
        float a, b, c;
        a = randFloat(-eps, eps);
        b = randFloat(-eps, eps);
        c = randFloat(-eps, eps);
        glm::vec3 temp = glm::vec3(a, b, c);
        p.color.push_back(temp + color);
        p.lifespan = lifespan;
    }
    return p;
}

```

创建函数中，为每个粒子分配位置，速度，以及颜色（其中randInt(a,b) 和 randFloat(a,b) 函数能均匀的获得a至b中的某个随机数）

而在主循环中，通过调用 updateParticle 和 updateMagic 函数更新粒子参数（引入随机）

以 updateParticle 为例

```

void updateMagic(glm::vec3 time, Magic &myma)
{
    for (int i = 0; i < myma.pos.size(); i++)
    {

```

```

        if (myma.lifespan == 0 || !collisionDetection(myma.pos[i], 0.01f) ||
myma.pos[i].y <= -1.0f)
{
    isMagic = false;
    isParticle = true;
    p = initParticle(myma.pos[0], 3000, 100, glm::vec3(0.8, 0, 0), 0.5);
    break;
}
myma.pos[i] = myma.pos[i] + myma.speed * time;
}
myma.lifespan--;
}

```

对于粒子来说，消失有两种情况，一是粒子的**lifespan**将为0，自动消散；二是粒子碰撞到物体上，需要经过**collisionDetection**的检测。每次执行这个函数，粒子的**lifespan**减一。如果Magic粒子消失，则产生一个Particle粒子，实现爆炸效果。

3.9 阴影贴图 (PCF)

阴影是光线被阻挡的结果，当一个光源的光线因被阻挡而不能达到一个物体表面时候，物体则在阴影中。本次大程我们采用阴影贴图 (Shadow mapping) 来进行设计。

具体想法是，以光的位置为视角进行渲染，在光照的位置获得深度贴图，利用投影矩阵和视图矩阵结合一起将任何三维位置转变为光源的可见坐标。如果物体的深度大于深度贴图中的深度，则能判定该点在阴影中。

而生成深度贴图，使用帧缓冲进行生成

```

const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
 glGenFramebuffers(1, &depthMapFBO);
// create depth texture
unsigned int depthMap;
 glGenTextures(1, &depthMap);
 glBindTexture(GL_TEXTURE_2D, depthMap);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH,
SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
 float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
 glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
 glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
depthMap, 0);
 glDrawBuffer(GL_NONE);
 glReadBuffer(GL_NONE);
 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

经学习发现，我们需要显示告诉帧缓冲我们不适用颜色缓冲，将调用glDrawBuffer和glReadBuffer把读和绘制缓冲设置为GL_NONE。

然后去获取深度贴图。

```

if (isShadow == true)
{

```

```

        glm::vec3 lightPos(-3.0f, -1.0f, -3.0f);
        float near_plane = 1.0f, far_plane = 7.5f;
        lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
far_plane);C6+
        lightView = glm::lookAt(camera.Position, camera.Position + camera.Front,
camera.Up);
        lightSpaceMatrix = lightProjection * lightView;
        DepthShader.use();
        DepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
        glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
        glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
        glClear(GL_DEPTH_BUFFER_BIT);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, diffuse);
        renderScene(DepthShader);
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    }
}

```

其中 `renderscene` 函数即为绘制迷宫地图等各个物体，在指定 `lightview` 光的视角、`lightProjection` 光投影矩阵，利用 `lightView*lightProjection` 可以将每个世界空间坐标变换到光源处所见到的那个空间，进而获得深度图。

使用的vs, fs文件如下：

```

#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}

#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
}

```

而后利用这个深度纹理图，绘制物体

```

glActiveTexture(GL_TEXTURE2);
	glBindTexture(GL_TEXTURE_2D, depthMap);

```

其中物体的fs文件中，添加这么一个函数

```

float ShadowCalculation(vec4 fragPosLightSpace)
{
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closestDepth = texture(material.shadowMap, projCoords.xy).r;
}

```

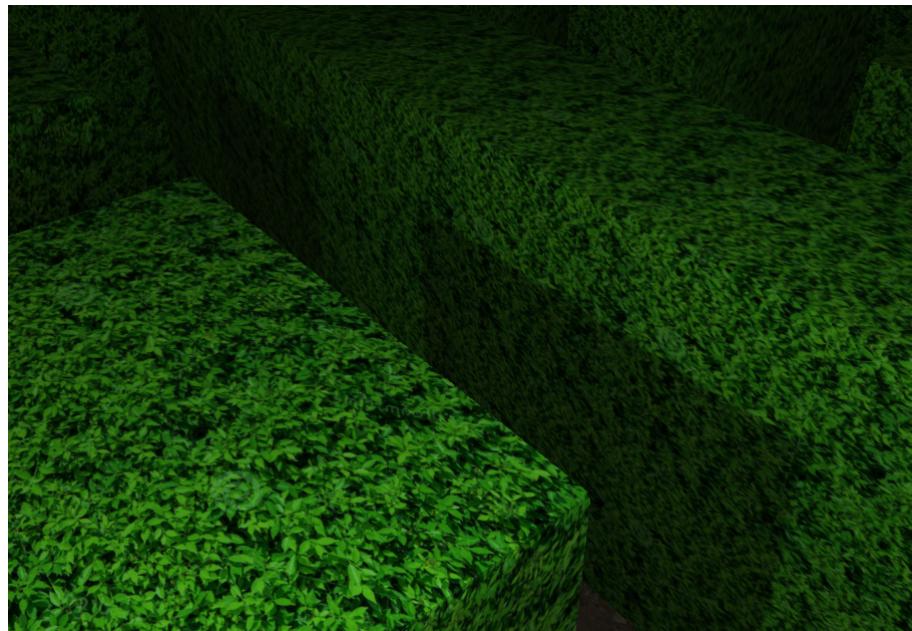
```

float currentDepth = projCoords.z;
vec3 normal = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(material.shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(material.shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
if(projCoords.z > 1.0)
    shadow = 0.0;

return shadow;
}

```

其中 `closestDepth` 是从深度图获得的深度，而 `currentDepth` 是当前点的到光的深度。而后我们采用阴影偏移解决 Shadow acne 问题。同时使用 PCF (percentage-closer filtering) 方法，使用多次采样，每次采样的纹理不同，然后进行平均化获得柔和阴影。利用 `shadow` 系数，借助 $(1 - \text{shadow}) * (\text{diffuse} + \text{specular})$ 获得阴影。



3.10 HDR&&Gamma 修正

HDR(High Dynamic Range, 高动态范围)可以让亮的东西可以变得非常亮，暗的东西可以变得非常暗，而且充满细节。自动在过暗和过亮的环境下调节。

我们也采用帧缓冲器来进行 HDR 的渲染。

```

unsigned int hdrFBO;
 glGenFramebuffers(1, &hdrFBO);
 unsigned int colorBuffer;
 glGenTextures(1, &colorBuffer);
 glBindTexture(GL_TEXTURE_2D, colorBuffer);

```

```

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0,
GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
unsigned int rboDepth;
 glGenRenderbuffers(1, &rboDepth);
 glBindRenderbuffer(GL_RENDERBUFFER, rboDepth);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, SCR_WIDTH,
SCR_HEIGHT);
 glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
colorBuffer, 0);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, rboDepth);
 if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "Framebuffer not complete!" << std::endl;
 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

HDR渲染所用的fs文件如下：

```

#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

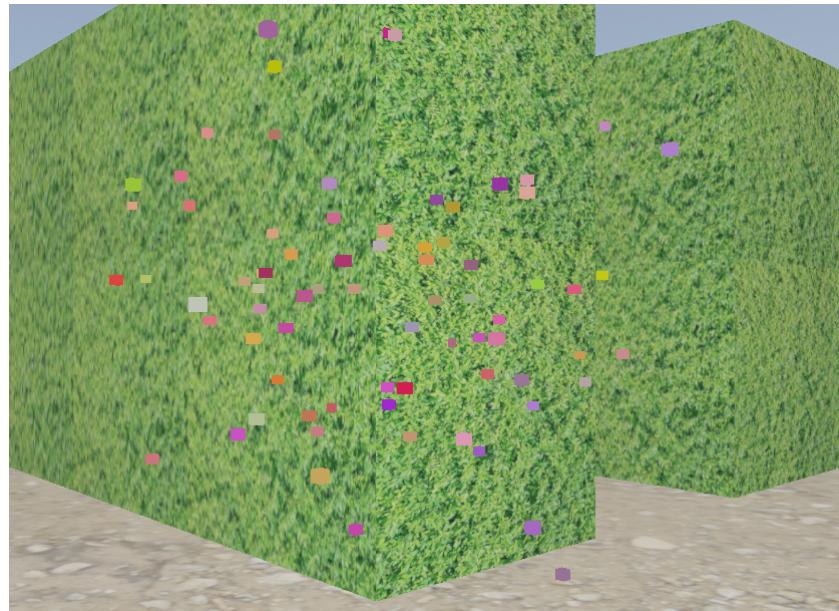
uniform sampler2D hdrBuffer;
uniform bool hdr;
uniform bool gm;
uniform float exposure;

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;
    if(hdr)
    {
        // reinhard
        vec3 result = hdrColor / (hdrColor + vec3(1.0));
        // exposure
        vec3 result = vec3(1.0) - exp(-hdrColor * exposure);
        if(gm)
        {
            result = pow(result, vec3(1.0 / gamma));
        }
        FragColor = vec4(result, 1.0);
    }
    else
    {
        if(gm)
        {
            vec3 result = pow(hdrColor, vec3(1.0 / gamma));
            FragColor = vec4(result, 1.0);
        }
        else
        {
            FragColor = vec4(hdrColor, 1.0);
        }
    }
}

```

```
    }  
}
```

实验中我们可以采用Reinhard色调映射来将HDR转化为LDR，最后再使用一个Gamma校正。效果如下：



3.11 怪物系统

在我们的游戏中增加了怪物系统，怪物由不同的**模型、三维信息、生命值、模型放缩比例、位置信息、旋转角度**等信息。由于我们使用的obj文件不太容易单独控制模型的手部执行攻击动作，所以在我们的设定下，怪物没有攻击方式，但是存在碰撞体积和生命值，并且怪物们会时刻**保持面向玩家**的方向起到恐吓的效果。

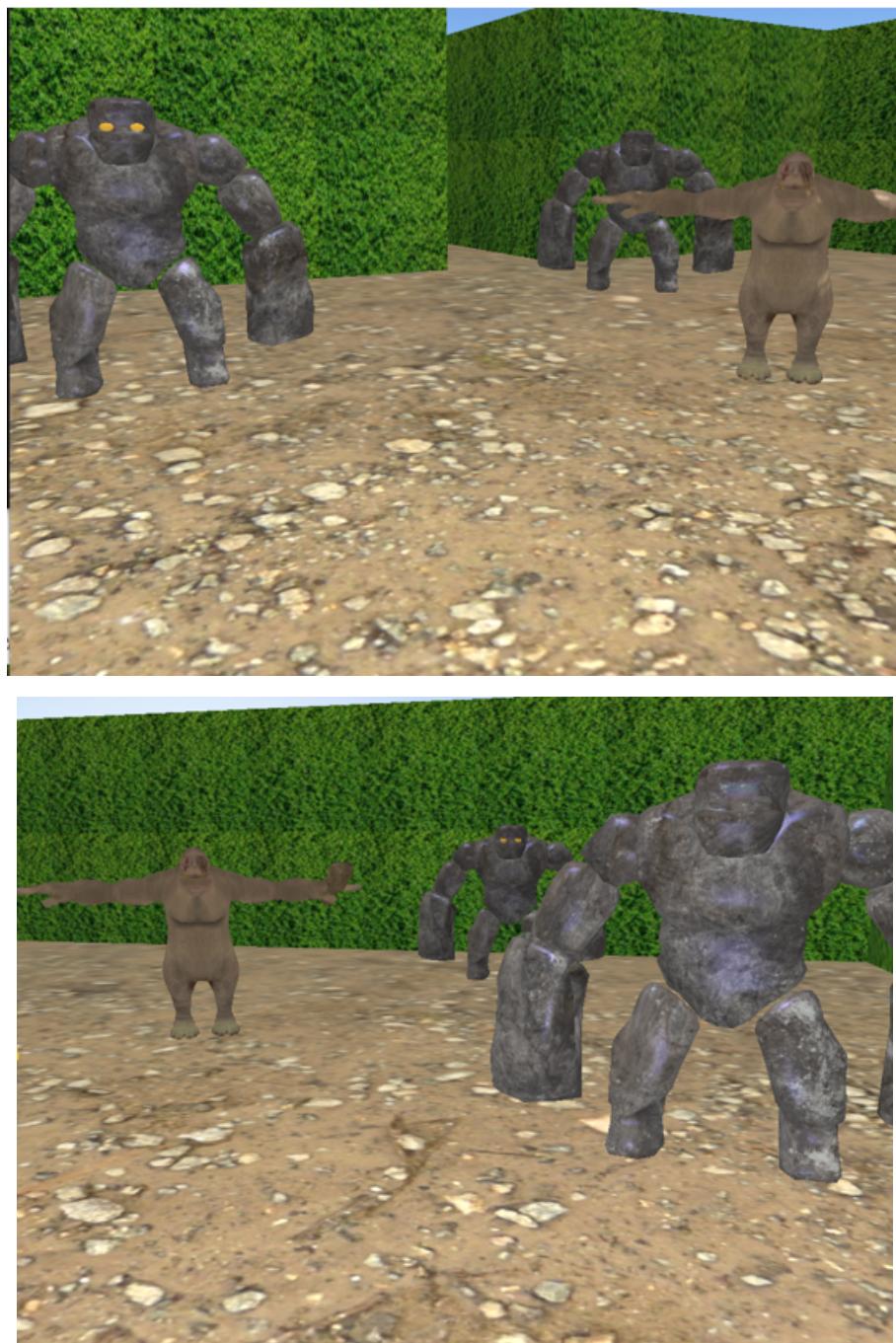
具体实现的monster类包含上述变量数据外，还有

- 受到攻击时更新生命值
- 时刻保持面向玩家位置
- 死亡消失

下面是Monster类的定义：

```
class Monster {  
public:  
    glm::vec3 position;  
    glm::vec3 volume;  
    glm::vec3 front;  
    bool isDead = false;  
    int HP;  
    float zoomRate = 1.0f;  
    float rotateAngle = 0.0f;  
    unsigned int type;  
    Monster();  
    Monster(unsigned int type, float zoom_rate, glm::vec3 volume, glm::vec3  
position, int HP);  
    ~Monster();  
    void UpdateHP(int damage);  
    void UpdateAngle(glm::vec3 targetPosition);  
private:  
    void dead();  
};
```

下面是几张怪物的图片：



四、工程文档结构

本项目的实现基于glfw库，首先在包含目录下需要有assimp库、GLFW库、glad库、glm库以及KHR库。



在库目录中需要有根据自身VS版本的assimp-vc140-mt.dll和assimp-vc140-mt.lib文件，以及glfw.lib文件。



而在项目文件夹下，包含使用到的库文件如下：

assimp-vc140-mt.dll	2021/1/6 22:30	应用程序扩展
assimp-vc140-mt.lib	2021/1/6 22:30	Object File Library
camera.h	2021/1/10 20:34	C/C++ Header
cg_project.vcxproj	2021/1/12 1:58	VC++ Project
cg_project.vcxproj.filters	2021/1/12 1:58	VC++ Project Fil...
cg_project.vcxproj.user	2020/12/31 15:22	Per-User Project...
filesystem.h	2021/1/6 22:45	C/C++ Header
global.h	2021/1/12 3:43	C/C++ Header
main.cpp	2021/1/12 4:52	C++ Source
mesh.h	2021/1/6 22:37	C/C++ Header
model.h	2021/1/6 22:12	C/C++ Header
monster.cpp	2021/1/12 4:09	C++ Source
monster.h	2021/1/12 2:24	C/C++ Header
root_directory.h	2021/1/7 20:58	C/C++ Header
shader.h	2020/12/11 14:37	C/C++ Header
shader_m.h	2020/12/11 14:37	C/C++ Header
shader_s.h	2020/12/11 14:37	C/C++ Header
stb_image.h	2020/12/11 14:37	C/C++ Header

使用到的纹理、模型、点着色器、片着色器等资源都存放在 resource 文件夹下，分为 obj , material , skybox , vs , fs 等不同文件的子目录。

五、结束语

我们一开始采用glut的opengl框架下实现，但因为有诸多限制，故在项目中间我们毅然决然的选择了GLFW框架来完成我们的项目。在完成项目过程中，我们通过LearnOpenGL学习到较为底层的实现方法（点着色器、片着色器）等，学习到PBR、延迟渲染、BRDF等高级光照的实现，并尝试将一些高级模型（如阴影贴图）等方法一个个添加到我们的项目中，受益匪浅。

小组成员：

黄仁泓 / 3180101969

李昊 / 3180105681

参考资料：

GLFW资源

<https://learnopengl.com>

模型来源

<https://free3d.com/>

<https://www.turbosquid.com/>

特别鸣谢

十分感谢王雨行同学帮忙绘制精美的UI界面和视频剪辑